SATOS: Storage Agnostic Tokens over Opaque and Substructural Types

Knecht, Markus ; Stiller, Burkhard

DOI: https://doi.org/10.1109/CVCBT.2019.000-4

SATOS: Storage Agnostic Tokens over Opaque and Substructural Types

Markus Knecht, Burkhard Stiller
Communication Systems Group CSG, Department of Informatics IfI, University of Zurich UZH
E-mail: markus.knecht2@uzh.ch
stiller@ifi.uzh.ch

***Abstract*—Public blockchains in support of Smart Contracts (SC), like Ethereum enable everyone to represent scarce, valuable resources (like cryptocurrencies) as so-called tokens. Token issuing and management was the first blockchain use case. However, programming languages and runtime systems used in the current blockchains for their SCs lack a secure and straightforward way to implement and handle tokens. The unnecessary complexity in doing so can lead to erroneous implementation of tokens and applications built on top of these, including the loss or theft of tokens as it happened. The most known attack was "TheDAO" attack which led to the "loss" of tokens, valued at that time at approximately 60 M US Dollar.**

A better and secure token representation directly embedded into a SC runtime and SC programming language could prevent loss of tokens. Thus this paper presents an approach including parts of a programming language using it. The core of the model is to use opaque and substructural data types together with an on-chain soundness checker to generically represent tokens securely as values similar to integers and booleans. Opaque data types enforce that only a designated piece of code can create values of that type. The substructural data types allow values to express scarcity by preventing the duplication and elimination of values. The on-chain soundness checker ensures that the deployed code does not violate guarantees given by the type system, which includes opaque and substructural data types.

Keywords—blockchain; smart contracts; programming languages; type systems; opaque data types; substructural data types; UTxO model; account model

I. Introduction

The concept of a Smart Contract (SC) was first presented in [30], but became practically viable with the invention of blockchains [26]. SC-enabled blockchains like Ethereum [15], [33] made them easy to use, especially, different concepts of distributed and untrusted stakeholder interactions, which had been unpractical before. One of these concepts is a tokenization, which enables the representation of existing or new properties, assets, and other scarce resources digitally on a blockchain. These tokens can be traded or managed by SCs without the need for an intermediary. The first blockchain token was the bitcoin token which is freely tradeable on the Bitcoin blockchain [26]. While the current Bitcoin blockchain does support scripts to specify under which conditions bitcoin tokens can be transferred, the Ethereum blockchain introduced full SCs, which allowed to run Turing-complete programs on the blockchain. The virtual machine of Etherum (EVM) executes SCs and is powerful enough to execute code defining and managing custom tokens.

With the power to use code to define new tokens and SCs operating on these tokens, comes the risk of erroneous code that can undermine the original intention. The fact that SC programming languages like Solidity [10] do not have features aimed at providing a secure and robust way to represent and manage tokens intensifies this. Some blockchains like WAVES [7] circumvent this problem by allowing participants to define new tokens without needing to code. That approach makes it easy to define secure tokens as long as these fit into the model dictated by the blockchain, however, it cannot be considered a general approach, since alternative representation can not be expressed.

Most blockchains with SC capabilities currently use the account model to store information. This determines an easier approach to adapt existing programming language concepts to describe SCs. A second approach is the Unspent transaction output (UTxO) one, where SCs determine conditions under which a token can be spent. This approach lacks a path to provide SC functionality that is powerful enough to introduce new tokens. Compared to the account model the UTxO model has inherent benefits like parallel transaction execution and better privacy guarantees, which may be minimal but in place [8]. Projects that introduce more powerful SC capabilities into the UTxO model do this either over a second layer separate from the primary blockchain [6] or by using a hybrid model that uses an account model for the SC part and an UTxO model for the transaction part [28].

Introducing SCs that can define new tokens on a UTxO blockchain without introducing a second layer or fall back to an account model is an open problem. Another related problem is how to represent tokens in a way that they are less prone to coding errors and easier to handle in an SC programming language. In this paper, we try to answer the more general research question on how to represent tokens and other scarce resources securely over SCs, such as their representation is independent of the storage and transaction handling model a blockchain uses and does not require an additional layer.

This paper defines, a new model, termed SATOS for representing and managing tokens on a blockchain as first-class values. This makes it easy to represent tokens in a SC language or virtual machine as they are represented similarly to other values like integers or booleans. Furthermore, the model is agnostic to the storage model and transaction handling methods. This approach can be embedded into account-based and UTxO-based blockchains. SATOS makes it easier to design and optimize the storage and transaction layer independent of the SC execution layer. The programming language called

978-1-7281-3669-1/19/$31.00 ©2019 IEEE
DOI 10.1109/CVCBT.2019.000-4

Mandala, currently under development, is used to provide examples visualizing the major concepts. While Mandala will address other relevant factors of SCs that can lead to vulnerabilities like access control management and error handling, this paper will focus on SATOS and its approach to managing tokens and other scarce resources. The best-known attack on a SC termed "TheDAO" attack could have been prevented in the SATOS model, because tokens are first class values and do not require separate balance tracking (which was out of sync in "TheDAO" attack) [31]

II. Token Representation Models

Two approaches exist today to provide SCs. Bitcoin uses a computationally less powerful approach, where the SC consists of code representing a partial script attached to tokens [13]. To spend tokens a transaction has to provide the missing parts of the script so that when executed it evaluates to true. All tokens spent in a transaction can be used to generate new tokens with different SCs attached. The sum of the tokens generated must be the same as the sum of the tokens spent. With this approach, the token scarcity and functionality is independent of the SCs. Thus new tokens can not be introduced by them. This has the UTxO model as its foundation: where each transaction produces outputs and each of them can be used as input to exactly one future transaction.

The computationally more powerful approach introduced by Ethereum creates a new storage region called an account on the blockchain (storing the contract state) for each SC instance [15]. The code associated with the SC can store and read arbitrary values to and from that region but cannot directly access the storage region of other SC instances. However, interaction is possible by executing the code associated with that instance which can interact with its storage and provides a result to the calling instance. Thus tokens can be expressed by a SC storing who owns how many tokens and providing functions to transfer tokens to another entity after the owner authorized the transfer. With this approach, a token's scarcity is determined by the code in the SC guarding its storage region. This SC model uses the account model as its foundation, where each entity (be it a SC or a account controlled over a asymmetric key pair) has an identity (called an address) on the blockchain. Tokens and other values can be associated with these identities to express ownership.

The contract storage region approach often leads to monolithic code that manages storage regions, as every allowed state transition in that storage region has to be expressed in a single SC. The monolithic SC has to cover all cases from the beginning as SC code deployed on a blockchain can no longer be modified [33]. In turn, it is hard to apply secure coding practices like abstraction, code reuse, or separation of concerns. Attempts to use these practices had lead to unexpected flaws, which lead to the loss of large amounts of valuable tokens [29]. Defining new tokens and corresponding functionality is prone to errors that can lead to financial loss. Associated risks on defining custom tokens are tackled today by code reviews, bug bounties, and test runs on test nets. These countermeasures are resource-and time-intensive and provide a substantial barrier for defining tokens and writing contracts operating on these tokens.

In contrast to existing approaches, in the SATOS model, the scarcity and functionality of new tokens can be enforced independently from the storage technique and location. A SC can introduce new value types and associated functions consuming and creating values of that type. Functions not associated with those types can still receive and return those values, but can not interact with them except over associated functions. This concept is called opaque data types and is frequently used to implement the more general concept of abstract data types [24].

Opaque data types can enforce the functionality of a token by restricting manipulations, but they fail to enforce the scarcity of a token. To address this shortcoming, type declarations restrict the number of usages of values of that type. A token type can specify that values must be used at most once and, thus, preventing a function from making a copy of that value. This concept is called an affine type and is part of a broader concept named substructural data types [32]. For representing tokens, affine types would be sufficient but by supporting other substructural data types like linear (must be used exactly once) and relevant (must be used at least once), further non-token related use cases are expressed more elegantly. SATOS is independent of the concrete storage concept used by the blockchain as long as a concrete implementation does enforce guarantees given by opaque and substructural data types. Figure 1 shows for values of a substructural as well as non-substructural (regular) type what happens between their creation and consumption. The creation and consumption can only happen inside a function associated with that type. Between the create and consume operation an arbitrary amount of time can pass.

Opaque and substructural data types can be enforced during compilation of a program or at runtime. Approaches that enforces these rules at runtime do impose a performance overhead during the execution, since the types of values have to be passed around (higher memory usage) and have to be enforced for each operation (higher CPU usage). Checking them at compile time reduces the total costs, since checks occur only once. Checking types only at compiletime is not sufficient for public blockchains, since this does not prevent an attacker from deploying code that was compiled by a manipulated compiler, thus, not producing sound programs. Using a respective compiler would allow an attacker to deploy code that could freely create and duplicate values, even if opaque and substructural data types would not allow for it. To prevent this problem, SATOS leverages a technique, where the soundness of the code is checked during the deployment to the blockchain and non-sound code is refused [23]. These soundness checks have to be performed only once and are part of the blockchain validation rules. Consequently, SATOS can enforce any static guarantee, not only for opaque and substructural data types. A particular embedding of SATOS into a blockchain does still require runtime checks at the start of a transaction's execution, because input values have an origin that is out of reach of the soundness checker running during code deployment. These inputs could come from the blockchain's storage or may be a part of the transaction itself. The checks during deployment help developers to provide correct code but do not free them from the responsibility of providing correct code.

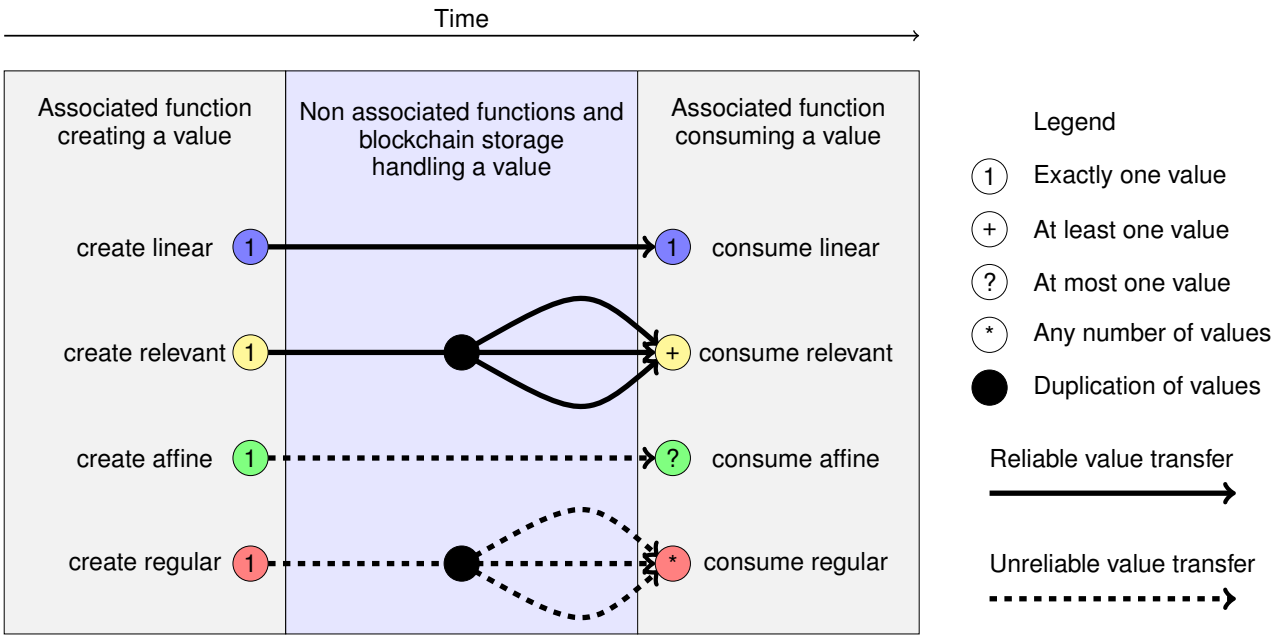

Figure 1. Lifetime of values with opaque and substructural data types

III. Related Work

There is a limited amount of research and projects that are related to SATOS. On the side of SC programming languages that use similar concepts as SATOS there is Obsidian [16]. Other non-SC programming languages that use opaque or substructural types exist, like Rust [5], but they do not use the type system to represent scarce ownable resources as these concepts are not of importance outside of SC's. In respect to improving the capabilities of UTxO based SC representations, there are a few non-academic projects [18], [27], [28].

A. Obsidian

The SC programming language Obsidian [16] is under development, where linear types are used as part of the representation of tokens. Obsidian focuses on the interaction of users with language features (user-centric design) and less on the integration of a language into blockchains. Obsidian applies an object-oriented paradigm combined with typestates [12] where every object represents a state machine. The compiler tracks the state of an object statically. One reference to the object is a "owned reference" which has a linear type to ensure that there are no aliases (other references pointing to the same object), enabling the representation of owned tokens. Checking the soundness of a program is performed by the classical approach, while these guarantees are enforced during off-chain compilation. This approach is sound in systems where all interacting SCs are deployed by entities trusting each other to use only the Obsidian compiler. Functions can only receive and return owned values from or to trusted callers as an untrusted caller may not enforce guarantees given by the ownership model. Obsidian is best suited for permissioned blockchains like Hyperledger Fabric [1] (current compilation target [17]). In permissioned blockchains the right to deploy SCs can be limited to a trusted entity, be it an individual, company, or a consortium with a governance process. In contrast, SATOS addresses and provides a solution to the runtime enforcement problem, which allows opaque and substructural data types as parameters and returns in functions called by untrusted SCs. Thus, SATOS is well suited for public blockchains, since no trust assumptions are necessary and anybody can deploy SCs.

B. Language Runtimes

Techniques to efficiently enforce soundness of code loaded into a runtime are a well-researched topic as most modern virtual machines like WASM [20] and the JVM [2] do this. These virtual machines operate on a byte code that can be checked efficiently, often in a single pass. The guarantees given by the byte code are often less then what a higher level language that compiles to it guarantees. Modern virtual machines check soundness constraints like type safety, function visibility and reference integrity (checking that a referred variable or function exist) amongst others.

In a non-blockchain execution environment, these virtual machines have to check the soundness each time they load code because it may have been modified on the disk since it was checked the last time. SATOS, on the other hand, runs on a blockchain which can guarantee that the code is not modified after deployment and thus it suffices to check the soundness once when the code is deployed.

C. UTxO SC Capabilities

Multiple non-academic projects [18], [27], [28] want to improve SC capabilities available to public UTxO-based

blockchains. Currently three prominent approaches exist to improve SC capabilities in the UTxO model:

1) Improving the expressiveness of spending conditions without addressing their inherent inability to introduce new tokens [27].
2) Providing hybrid models that use the UTxO model for transactions and the account model for SCs and provide interaction capabilities between them [18].
3) Adding specific mechanisms for defining new tokens with a fixed behavior [28].

The approaches 1. and 3. are less expressive than what an account-based SC platform can express. The hybrid model 2. has the limitation that an UTxO transaction interacting with an SC has to follow the restrictions from both models, eliminating advantages that are exclusive to one of these two models. In contrast, SATOS can describe the introduction of new tokens in an UTxO model without limiting the functionality of the token or eliminating the benefits of the UTxO model.

D. Orthogonal Approaches

Providing security oriented programming languages with a well defined operative semantic is just one approach to reduce bugs and eliminate attack vectors in SCs. Many other orthogonal approaches can be used to achieve the same goal and even approaches not specifically targeting SCs can often be applied to some degree in a blockchain environment. Orthogonal approaches include but are not limited to: Formal verification, code review, bug bounties and testing. Some noteworthy approaches that have SCs in mind are: Oyente [25], which provides a static analysis framework to find vulnerabilities in SCs, the Hydra Framework [14] which automates bug bounty payouts and the KEVM [21] which provides a formal specification of the EVM that allows to do formal verification on SCs. This list is not complete as the field is vast, and there is much impact-full research.

IV. Token Implementation with Opaque and Substructural Data Types

This papers uses Mandala, an under development SC programming language to show how SATOS is used in practice to create tokens and manage them. SATOS unrelated Mandala concepts are required when implementing tokens as opaque and substructural data types do not work in isolation. Mandala is a statically typed domain-specific smart contract language based on side-effect free statically dispatched functions and without recursion or loops. These properties make Mandala a not Turing complete programming language, but other languages based on SATOS can be Turing complete. This paper will not provide an exhaustive description of Mandala but instead, focus on how SATOS is used in Mandala to represent tokens.

SATOS requires that there is a way to associate functions to types with the purpose to ensure that only these functions can construct or deconstruct values of the type. Mandala achieves this over its module component which groups together a set of functions and a set of types. Simultaneously modules are the smallest deployment unit and provide a namespace for its types and functions allowing other modules to refer to them. Mandala's data types are algebraic data types (ADT), and Mandala has a parametric polymorphic type system. ADT's are used in functional languages like Haskell [22] and are composite types, that can contain other values. An ADT is an immutable data type where a created value cannot be modified after construction. Parametric polymorphism allows data types and functions to take types as parameters (generic functions and generic data types). Values constructed from the same data types but with different type parameters have a different type and are assignment incompatible.

A sound interaction between, substructural data types, and parameter polymorphism with ADT's requires them to declare a substructural category which can be linear, affine, relevant or regular where regular indicates that the ADT is not substructural and the others are the substructural options. These categories are hierarchical concerning the guarantees they provide. A linear ADT provides stronger guarantees as an affine or relevant ADT, and these provide stronger guarantees as a regular ADT. Types have a substructural category as well which is the weakest category that is at least as strong as all the categories of the applied type parameters and the category of the ADT declaration. As an example, an affine ADT with one type parameter instantiated with a relevant or linear type would have the linear category. Field types in an ADT cannot have a category with stronger guarantees than the declared category on the ADT itself. Type parameters of an ADT are treated as if they have the regular category when used to construct the types of fields.

Function declarations in Mandala can mark its type parameters as protected. A caller can only apply types for the protected type parameters where the corresponding data type definition is in the same module as the calling function. The purpose of protected functions is to provide further capabilities to opaque data types which are needed when defining tokens as can be seen in the code examples of this section.

Mandala's syntax is inspired by functional languages and uses pattern matching for deconstructing values and let bindings for naming results of expressions. There are different classes of tokens on top of blockchains. The most fundamental differentiation is between fungible and non-fungible tokens (For Ethereum specified in ERC-20 [3] and ERC-721 [4]), which both can be represented in SATOS. The remainder of this section will focus on fungible token and show all that is necessary to provide and manage them including ownership management and minting. Fungible tokens have the characteristic that all tokens are interchangeable, and it is not important which specific tokens somebody owns.

A. Token Definition

The storage agnostic nature of SATOS allows a high level of abstraction, reusability and separation of concerns, allowing a generic token description that can be reused to represent different tokens. Functionality that in other smart contract languages like solidity [10] would need to be part of the token description can in Mandala be part of separate modules. Listing 1 shows a reusable fungible token definition with a minimal set of functionalities. In languages based on SATOS such a module can be part of the standard library to set a standard allowing different applications to speak a common

language where fungible tokens are concerned. In ethereum and solidity a standard called ERC-20 had to be created to enable a common representation of fungible tokens.

On line 2 in Listing 1 an affine and opaque ADT named *Token*, representing the token type is declared. The value of that type tracks the number of fungible tokens it represents as an unsigned 128bit integer. The type parameter *T* represents the particular token and makes the ADT reusable. As an example, the types *Token[Btc]* and *Token[Eth]* are different types representing different tokens. Over the *mint* function on line 4 new tokens can be generated. The type parameter of *mint* is protected which ensures that only the module defining the type applied to *T* can call it, preventing unauthorized minting. The functions *merge* and *split*, on the lines 9 and 17, allow to rearrange the grouping of token quantities. These functions have the keyword risky indicating that they can produce an error which is the case because Mandala produces an error on an integer over-or underflow.

Listing 1. Generic Token Module

```
module Token {                                          1
  affine type Token[T](u128)                            2
                                                        3
  protected[T] function                                 4
  mint[T](a:u128) {                                     5
    return Token[T](a)                                  6
  }                                                     7
                                                        8
  public risky function                                 9
  split[T](Token[T](a), split:u128){                    10
    return (                                            11
      Token[T](a-split),                                12
      Token[T](split)                                   13
    )                                                   14
  }                                                     15
                                                        16
  public risky function                                 17
  merge[T](Token[T](a1), Token[T](a2)){                 18
    return Token[T](a1+a2)                              19
  }                                                     20
                                                        21
  public function                                       22
  balance[T](Token[T](val)) {                           23
    return (val,Token[T](val))                          24
  }                                                     25
}                                                       26
```

The token representation from Listing 1 has multiple benefits compared to the previous approaches that do not use SATOS for token representation.

1) Two different tokens cannot be assigned to each other, preventing accidental mix-ups like sending ETH tokens when BTC tokens where expected
2) Tokens can be passed around like any other value without the need of delegating this to another smart contract
3) Tokens are generic and can be used without knowing details about the token
4) Function signatures do explicitly state which tokens they expect and which tokens they return
5) The Token implementation can be reused instead of being re-implemented for each new token

B. Blockchain Integration

In SATOS values are agnostic to the blockchain storage and do not care where and how they are stored. A blockchain using SATOS has to provide one or more modules containing ADTs that capture the values to store and the necessary meta information, as well as the functions that operate on the values of these ADTs. Listing 2 shows a generic example for a blockchain using the UTxO model. In the presented example Mandala can be used to express the spending condition as well as expressing the transaction validity rules. This representation is not Bitcoin specific and it is enough to understand the UTxO principles to apply it.

Listing 2. UTxO Module

```
module UTxO {                                           1
  top linear type UTxO[T](T)                            2
                                                        3
  public function create[T](t:T){                       4
    return UTxO[T](t)                                   5
  }                                                     6
                                                        7
  protected[T] function                                 8
  spend[T](UTxO[T](val)) {                              9
    return val                                          10
  }                                                     11
}                                                       12
```

The *UTxO* data types on line 2 in Listing 2 is marked with the top keyword indicating that it is not allowed to use values of that ADT as constructor arguments. This is necessary to prevent an attacker from taking an *UTxO* that he is not able to spend and embed it in an *UTxO* that only he can spend. To limit how a *UTxO* can be spent the generic parameter *T* of the spend function on line 10 is protected allowing the module defining *T* to enforce the spending condition.

Listing 3. account Module

```
module account {                                        1
  top linear type AccountEntry[T](                      2
    data20, data20, T                                   3
  )                                                     4
                                                        5
  public function create[T](                            6
    id:data20, key:data20,  t:T                         7
  ){                                                    8
    return AccountEntry[T](id,key,t)                    9
  }                                                     10
                                                        11
  protected[T] function                                 12
  spend[T](AccountEntry[T](id,key,t)) {                 13
    return (id,key,t)                                   14
  }                                                     15
}                                                       16
```

The account version in Listing 3 has a similar structure as the UTxO version, but the *AccountEntry* type on line 2 in Listing 3 additionally captures the location where to store the value. In the example, the storage location is represented as two 20 Byte (output size of a 160bit hash function) long values representing an account and a key pointing to a slot in the accounts storage region.

A blockchain that uses SATOS in junction with a SC language like Mandala has to provide a way for their transactions to call functions in modules. Beside the name of the function(s) that a transaction calls the parameters have to be specified which can be primitives (like integers) and the storage-specific types of the blockchain (like UTxOs). An advanced implementation will further allow calling multiple functions where the return values of one function are used as the parameter of another and in that case these values can have any type. The most elaborated integration can contain interpreted Mandala code extended by some primitives to exchange values with the blockchains storage. Independent on how the integration looks like a SC language using SATOS needs to provide signatures for its functions and types with enough information such that the transaction handler can make the necessary checks to enforce soundness when calling a function.

As functions in SATOS can not directly interact with the blockchains storage, they must be stateless, and only the blockchain, calling a function can load or store values. The data types provided by the blockchain can influence the functionality used to instantiate and transfer tokens. The UTxO or AccountEntry data types are not enough for many use cases. One data type that most blockchain integrations will provide is a top regular data type which contains different information about the current state like the block number or the executed transactions hash. Later examples assume that a such a data type called Context is provided. A value of the Context type never lives longer as the current transaction execution because the top keyword prevents its usage in an ADT that could be persisted.

C. Spending Conditions

Over the protected mint function from Listing 1 scarcity of a token can be controlled. For a token to develop value it is not enough to be scarce, a notion of ownership is required as well. Blockchains achieve this with the help of public key cryptography. Listing 4 extends the UTxO example with the implementation of a spending condition using public key cryptography. It is assumed that an ECDSA module is provided by the blockchain (similar to how Context is provided), that contains the primitives needed to verify ECDSA signatures.

Ownership is represented over the *Lock* ADT on Line 4 in Listing 4 which contains a public key (the owner) and a value (the property). While everyone can generate an *UTxO* that contains an owned value over the *lock* function on Line 6, the *unlock* function on Line 10 ensures that spending the value requires a valid signature over the transaction hash created by the owner. If the signature is not correct the require statement on line 18 will produce an error. The *unlock* function is the only way to retrieve the property as only functions from the *SigLock* module can call the protected *spend* function of a *UTxO* containing their *Lock*.

Listing 4. ECDSA based ownership

```
import ECDSA.*; import UTxO.*                          1
import Context.*                                       2
module SigLock {                                       3
  affine type Lock[T](Pk,T)                            4
                                                       5
  public function lock[T](pk:Pk, v:T){                 6
    return UTxO[Lock[T]](Lock[T](pk,v))                7
  }                                                    8
                                                       9
  public risky function unlock[T](                     10
    ctx:Context,                                       11
    uxto:UTxO[Lock[T]],                                12
    sig:Sig                                            13
  ){                                                   14
    let lock = spend[Lock[T]](UTxO)                    15
    let Lock[T](pk,v) = lock                           16
    let txtHash = getTxtHash(ctx)                      17
    require verify(pk,txtHash,sig)                     18
    return v                                           19
  }                                                    20
}                                                      21
```

D. Creating a Token

One core part when creating a new kind of tokens are the rules that dictate how new tokens are minted. This part is different for each specific token, whereas the other presented modules can be reused. Listing 5 shows how a simple initial coin offering(ICO) which allows buying new tokens over another token during a time window of 1000 blocks with an exchange rate of 1.

Listing 5. Token offered over ICO

```
import ECDSA.*; import UTxO.*                          1
import Token.*; import Context.*                       2
import SigLock.*                                       3
module MyICO {                                         4
  type MyToken                                         5
  const start:u64 = 1000                               6
  const end:u64 = 2000                                 7
  const myPk = createPk(0xe2...5f)                     8
                                                       9
  public risky function                                10
  buy(ctx:Context, pay:Token[Btc]){                    11
    let blockNo = getBlockNo(ctx)                      12
    require blockNo >= start                           13
    require blockNo < end                              14
    let (val, pay) = balance(pay)                      15
    return (                                           16
      mint[MyToken](val),                              17
      lock[Token[Btc]](myPk,pay)                       18
    )                                                  19
  }                                                    20
}                                                      21
```

The *MyToken* ADT on line 5 in Listing 5 has no constructor and thus cannot be created and is used as type parameter to the Token type from Listing 1. The *buy* function on line 10 accepts *BTC* (as *Token[BTC]*) tokens as payment and then mints an equal amount of *MyToken* tokens (as *Token[MyToken]*) on line 17. It uses the *blockNo* from the context to check that the time window is still open. The payment is sent to a predefined public key by using the *Lock* from Listing 4. The *buy* function expects plain tokens as payment, and locked tokens first have to be unlocked before used as a parameter to the *buy* function. This approach keeps the ICO generic concerning the ownership mechanism used to protect the payment. If the used blockchain can call multiple functions per transaction, then a transaction can first call an unlock function and then pass the resulting token to the buy function. In case the blockchain only supports a call to one function per transaction an integration function like the one in Listing 6 is necessary to make the example work.

Listing 6. Buy transaction function

```
import ECDSA.*; import UTxO.*                    1
import Token.*; import Context.*                 2
module ICOBridge {                               3
  public risky function                          4
  buy(                                           5
    pay:UTxO[Lock[Token[Btc]]],                  6
    ctx:Context,                                 7
    sig:Sig                                      8
  ){                                             9
    let tok = unlock(ctx,pay,sig)                10
    return buy(ctx,tok)                          11
  }                                              12
}                                                13
```

E. Transferring a Token

Listing 7. Transfer transaction function

```
import ECDSA.*; import UTxO.*                    1
import Token.*; import Context.*                 2
module TransferBridge {                          3
  public risky function                          4
  transfer[T](                                   5
    ctx:Context, amount:u128,                    6
    in:UTxO[Lock[Token[T]]],                     7
    sig:Sig, payPk:Pk, myPk:Pk,                  8
  ){                                             9
    let tok = unlock(ctx,in,sig)                 10
    let (keep,pay) = split(tok, amount)          11
    return (                                     12
      lock[Token[T]](payPk,pay),                 13
      lock[Token[T]](myPk,keep)                  14
    )                                            15
  }                                              16
}                                                17
```

In blockchains that allow calling more than one function and route returns of one to the parameters of another a token transfer can be realised without needing additional code. A token can be transferred by first unlocking the locked inputs with a signature of the owner than calling split and merge on the tokens to produce the desired partitioning of output tokens which are finally locked again with the public keys of the recipients. If the blockchain does not support the call of multiple functions then the transfer has to be expressed in a Mandala function like the one in Listing 7 but this approach is less flexible as it can only handle a fixed amount of inputs and outputs.

V. Enforcing soundness

The values for representing tokens are not the only part needed to make SATOS work. The second essential element is the code and the its sound execution at runtime. This approach assumes that the smallest deployment unit is a list of types and associated functions like the module in Mandala, referring to such a deployment unit as a module, which is content-addressable over the hash of its content. When a type or function depends on a type or function from another module, it refers to it by the module's hash and the position/index in the target module.

An initial and naive design is stateless and stores no information about the code on the blockchain. Thus, every transaction has to provide modules containing called functions and all modules they directly or indirectly depend on. All these modules have to be checked for soundness. Modules with the same content, but provided by different transactions represent the same module. This approach repeats soundness checks for each transaction and the transaction sizes grows due to all the code provided. However, it demonstrates that the code needed to execute a transaction can be reconstructed at any time as long as the code of the modules needed is available off-chain. All code-related information stored on the blockchain determines a performance improvement and provides a trade-off between bandwidth and CPU cycles against used disk storage.

A first improvement records hashes of all sound modules on the blockchain and runs the soundness check only the first time the code of a module is needed. Transactions still contain the code of modules. In turn, storing the content of sound modules in addition to the hash leads to a bandwidth optimization as transactions only have to include module hashes but no longer the code. Module content that is stored on the blockchain can be compressed by removing all information that is only required for checking the soundness but not for executing the code. Hybrid variants are possible, where rarely used modules are removed from the storage and redeployed again when needed.

Independent of which soundness check results are stored and which are recomputed, ensuring soundness when code is deployed is not free and checks have to be executed by every full node in the blockchain. It is important to note that existing SC virtual machines like the EVM or WASM based SC virtual machines [9], [11] have to enforce soundness as well, but compared with SATOS they need to enforce less constraints. While the EVM enforces the soundness when code is executed, WASM based virtual machines do a single pass over the whole code before it is executed to ensure that it is sound. More complex soundness checks can still be done efficiently as modern virtual machines like the JVM have proven.

The fact that no on-chain state is required for the code brings the benefit for longtime evolutions of a blockchain: (a) If the same module is deployed on multiple blockchains or multiple shards of a single blockchain, they provide precisely the same functions and types, (b) If two blockchains/shards have a way to transfer a value from one to the other, then it can immediately be used at its new place, and (c) this is fully compatible with computational proof systems [19] which allow for the generation of a proof that a module is sound and the blockchain only needs to verify the proof instead of the full module. Because functions of a language built with SATOS are side-effect free a computational proof system can be used for proofing the soundness of modules and for proofing outputs of an transaction execution.

VI. Discussion

One of the advantages of SATOS is that it enables more complex SCs in the UTxO model. One of the benefits of the UTxO model is that it is easy to detect if two transactions are independent of each other and as a result can be executed in parallel. If a transaction uses the output of another transaction as input, then parallel execution is not allowed. In the most implementation of the account model, it is more complicated to decide if two transactions are independent of each other as it is necessary to knwo what state is accessed by each transaction. SATOS, requires that all state dependencies are made explicit even when used in combination with an account model and thus it is easy to decide if two transactions can be executed in parallel.

Not all SCs can be written in a way such as the transactions can be executed in parallel. Which categories of SCs can be represented in a way such as parallel execution becomes possible will require more research. SATOS immutable values encourage developers to split their state into smaller pieces, which has the potential to increase the number of parallel executable transactions. Parallel execution could further be improved if the blockchain provides specific types (in addition to UTxO and AccountEntry, etc.) like atomic accumulators that are built with efficient concurrent write access in mind.

SATOS introduces a new programming paradigm. SATOS paradigm is close to the functional programming paradigm as it shares the focus on immutable datatype and side-effect free functions. On top of that SATOS introduces the concept of using opaque and substructural types to represent values that can represent scarce resources. This paradigm shift implies that creating programs has to be approached differently when using SATOS, similar to how writing a program has to be approached differently in an imperative language compared to a functional language. This bears the risk that a developer may struggle with writing complex SCs in Mandala or other SATOS based languages until they get used to the new paradigm.

The benefits of this paradigm shift is not that Mandala (and other SATOS based languages) allows to implement SCs that are not possible in other SC languages but instead allows to write SCs in a more robust, more reusable and less error-prone way similar to the benefits that functional programming provided over imperative and procedural programming. Further SATOS independence on the storage and transaction execution model of the blockchain allows to create SCs is blockchains that do not use the account model like UTxO based blockchains.

VII. Conclusions

Creating and managing tokens is still the primary use case for SCs, but SC-enabled blokchains like Ethereum and their associated virtual machine and programming languages do only have support for a native cryptocurrency (like ether). New tokens need to be created and managed by monolithic SCs, tracking the possessions of each entity interacting with the token and which provide functionality to owners to transfer tokens to other entities.

This paper provides the new model termed SATOS, where tokens can be presented as values that leverage opaque and substructural data types. In a programming language using SATOS, tokens are as easy to handle as other native data types like integers. No complex additional logic is needed to represent them. The focus on abstraction and reusability provided by SATOS allows tokens to be defined generically and securely once. They are reusable by each concrete custom token. SATOS is agnostic on how a blockchain handles transactions and stores values.

For evaluating SATOS an analysis of examples is used. A series of examples written in the programming language Mandala applying SATOS has shown that the concepts of opaque and substructural data types are capable of defining tokens, storing them on an UTxO-or account-based blockchain, and managing their ownership with asymmetric cryptography. The process to deploy, manage, and host code for a virtual machine supporting SATOS is defined in a flexible way such that additional concepts, like sharding, sidechains, and stateless blockchains can be leveraged.

SATOS is the first approach that enables SCs that can define and manage new tokens and other assets in a pure UTxO model. Other related work either add an account model on top of the UTxO model or use a non-general approach like requiring new tokens or assets to select from a fixed predefined set of classes which are then treated specially during runtime.

Compared to the SC programming language Obsidian which use similar concepts to represent tokens as native data types, SATOS has the advantage that it is compatible with a public blockchain where different untrusted entities deploy SCs. In such an environment Obsidian SCs would not be able to exchange tokens with other untrusted SCs securely.

References

[1] Hyperlegder fabric. Accessed: 2019-02-13. [Online]. Available: https://www.hyperledger.org/projects/fabric

[2] Jvm. [Online]. Available: https://docs.oracle.com/javase/specs

[3] (2015) EIP 20: ERC-20 Token Standard. [Online]. Available: https://eips.ethereum.org/EIPS/eip-20

[4] (2015) EIP 721: ERC-721 Non-Fungible Token Standard. [Online]. Available: https://eips.ethereum.org/EIPS/eip-721

[5] (2017) The rust reference. Accessed: 2019-03-20. [Online]. Available: https://doc.rust-lang.org/1.0.0/reference.html

[6] (2018) Counterparty extends bitcoin in new and powerful ways. Accessed: 2019-02-13. [Online]. Available: https://counterparty.io/

[7] (2018) Waves blockchain. Accessed: 2019-03-20. [Online]. Available: https://wavesplatform.com/products-blockchain

[8] (2019) Ethereum design rationale: Accounts and not utxos. Accessed: 2019-03-20. [Online]. Available: https://github.com/ethereum/wiki/wiki/Design-Rationale#accounts-and-not-utxos

[9] (2019) Ethereum flavored webassembly (ewasm). Accessed: 2019-03-20. [Online]. Available: https://github.com/ewasm/design

[10] (2019) Solidity documentation. Accessed: 2019-03-20. [Online]. Available: https://media.readthedocs.org/pdf/solidity/latest/solidity.pdf

[11] (2019) Webassembly parity vm design. Accessed: 2019-05-18. [Online]. Available: https://wiki.parity.io/WebAssembly-Design

[12] J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks, "Typestate-oriented programming," in *The 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '09. New York, NY, USA: ACM, 2009, pp. 1015–1022. [Online]. Available: http://doi.acm.org/10.1145/1639950.1640073

[13] G. Andresen. (2012) Pay to script hash. Accessed: 2019-04-10. [Online]. Available: https://github.com/bitcoin/bips/blob/master/bip-0016.mediawiki

[14] L. Breidenbach, P. Daian, F. Tramèr, and A. Juels, "Enter the hydra: Towards principled bug bounties and exploit-resistant smart contracts," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.* USENIX Association, 2018, pp. 1335–1352.

[15] V. Buterin, "Ethereum: A next-generation smart contract and decentralized application platform," 2014, accessed: 2019-03-20. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper

[16] M. Coblenz, "Obsidian: A safer blockchain programming language," in *The 39th International Conference on Software Engineering Companion*, ser. ICSE-C '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 97–99. [Online]. Available: https://doi.org/10.1109/ICSE-C.2017.150

[17] ——, "Obsidian language development," 2019, accessed: 2019-03-20. [Online]. Available: https://github.com/mcoblenz/Obsidian

[18] P. Dai, N. Mahi, J. Earls, and A. Norta, "Smart-contract value-transfer protocols on a distributed mobile application platform," 2018, accessed: 2019-03-20. [Online]. Available: https://qtum.org/user/pages/01.home/Qtum whitepaper_en v0.7.pdf

[19] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof-systems," in *The Seventeenth Annual ACM Symposium on Theory of Computing*, ser. STOC '85. New York, NY, USA: ACM, 1985, pp. 291–304. [Online]. Available: http://doi.acm.org/10.1145/22145.22178

[20] W. C. Group. (2019) Webassembly specification. Accessed: 2019-03-20. [Online]. Available: https://webassembly.github.io/spec/core/_download/WebAssembly.pdf/

[21] E. Hildenbrandt, M. Saxena, X. Zhu, N. Rodrigues, P. Daian, D. Guth, B. Moore, Y. Zhang, D. Park, A. Ştefănescu, and G. Roşu, "Kevm: A complete semantics of the ethereum virtual machine," in *2018 IEEE 31st Computer Security Foundations Symposium*. IEEE, 2018, pp. 204–217.

[22] S. P. Jones. (2003) The haskell 98 language and libraries: The revised report.

[23] M. Knecht and B. Stiller, "Smartdemap: A smart contract deployment and management platform," in *Security of Networks and Services in an All-Connected World*, ser. AIMS 2017. Springer International Publishing, 2017, pp. 159–164. [Online]. Available: http://www.aims-conference.org/2017/10.1007_978-3-319-60774-0.pdf

[24] B. Liskov and S. Zilles, "Programming with abstract data types," *ACM Sigplan Notices*, vol. 9, no. 4, 1974.

[25] L. Loi, C. Duc-Hiep, O. Hrishi, S. Prateek, and H. Aquinas, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 254–269. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978309

[26] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2009. [Online]. Available: http://bitcoin.org/bitcoin.pdf

[27] D. Pigeon. (2018) Crypto-conditions: Utxo-based smart contracts on komodo platform. Accessed: 2019-02-13. [Online]. Available: https://komodoplatform.com/crypto-conditions-utxo-based-smart-contracts/

[28] K. Seo, "Codechain: an end-to-end assettokenization system," 2018, accessed: 2019-03-20. [Online]. Available: https://codechain.io/CodeChain_white_paper_v0.1.0.pdf

[29] S. Shankar. (2017) Parity multi-sig wallets funds frozen (explained). Accessed: 2019-02-13. [Online]. Available: https://blog.springrole.com/parity-multi-sig-wallets-funds-frozen-explained-768ac072763c

[30] N. Szabo, "Formalizing and securing relationships on public networks," *First Monday*, vol. 2, no. 9, 1997.

[31] P. Vessenes. Deconstructing thedao attack; a brief code tour. Accessed: 2019-05-22. [Online]. Available: http://vessenes.com/deconstructing-thedao-attack-a-brief-code-tour

[32] D. Walker, "Substructural type systems," in *Advanced Topics in Types and Programming Languages*, B. Pierce, Ed. MIT Press, 2002, pp. 3–43.

[33] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," 2015, accessed: 2019-03-20. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf